The C++ Standards Committee: Progress & Plans

February 17, 2004

Walter E. Brown Marc F. Paterno

Computing Division

Fermi National Accelerator Laboratory

Motivation for this talk

- C++ has become the *lingua franca* for HEP computer programming:
  - But the scientific community is still under-represented in the C++ standardization effort
  - Fermilab joined the standards committee in 2000:
    - FNAL has full voting privileges
    - We are FNAL's designated representatives
- Our goal is to keep you informed:
  - Share our experiences and insights
  - Communicate developments re future C++
  - Solicit feedback for the committee

Overview

- Background information:
  - National & international umbrella organizations:
    - Internal committee structure & procedures
    - Formal and informal working arrangements
  - C++ standardization timeline
  - Work completed since: DRs, TC1, TR
- Ongoing work in language & library evolution:
  - DRs and TR …
  - … as prelude to C++0x

ISO JTC1-SC22/WG21

- ISO: International Standards Organization
  - JTC1: Joint Technical Committee for Information Technology
  - SC22: Subcommittee for Programming Languages, their Environments, and System Software Interfaces
  - WG21: Working Group for C++
- ISO membership:
  - Open only to national standards bodies …
  - … of which ANSI is one

ANSI NCITS/J16

- ANSI: American National Standards Institute
  - NCITS: National Committee for Information Technology Standards (formerly: Accredited Standards Committee X3)
  - J16: Technical Committee for Programming Language C++
- Fermilab is a voting member of J16

Working arrangements

- All meetings of WG21 and J16 are co-located:
  - 2x/year; one in North America, one international
- All formal votes are taken twice:
  - J16 first, with only its (U.S.) members voting
  - WG21 second, with only national bodies voting
- Informal consensus is reached
  before formal motions are brought to a vote:
  - Hence formal motions generally pass with no significant opposition
  - All members share a strong commitment
    to cooperation

Internal organization

- All meeting attendees work closely together for the common goal:
  - J16 and WG21
  - Voting “members” and non-voting “observers”
  - Famous/notorious and unknowns
- Four “working groups” (subcommittees):
  - Core language (25 pre-9/11; lately ~15)
  - Library (30+ pre-9/11; lately ~20)
  - Performance (<10 and diminishing)
  - Evolution (~15 and growing)

C++ standardization timeline

- ~1990: beginning of standardization effort
- '95, '96: C++ Draft Standards issued for public comment; concerns addressed
- '97: Final C++ Standard approved
- '98: ISO balloting completed and ratified; 14882:1998 (informally: C++98) issued
- “1997-2000 was a deliberate period of calm to enhance stability” (B. Stroustrup)

1998-2003 accomplishments

- DRs (Defect Reports):
  - Apparent error, inconsistency, ambiguity, or omission in the published final Standard
  - Failure of wording to meet Committee's intent
- TC (Technical Corrigendum) #1:
  - Collection of corrections to accepted DRs
  - Merged with Standard, yielding ISO 14882:2003
  - BSI authorized book publication (Wiley, 2003)
- TR (Technical Report) on C++ Performance:
  - ISO balloting now in progress
  - Approval, issuance expected shortly

Sample Defect Report

- Library Issue 69:
  "Must elements of a **vector** be contiguous?"
  - Affects *Clause 23.2.4*
  - Status: DR (an accepted defect with an agreed resolution); part of TC1
  - Resolution: "The elements of a vector are stored contiguously…"
- Few issues were/are this straightforward

2001 to date

- 2001: *Directions for C++0x* seeded committee discussion re Standard C++ future
  - LWG began work toward a *Technical Report on C++ Library Extensions*
  - Full Committee to vote on final draft in late 2004
- 2002: formally decided to revise the Standard
  - ISO requirement: must decide every 5 years to ratify, amend, or withdraw
- All work now effectively aimed at C++0x:
  - Incorporate post-TC1 corrections & the LWG TR
  - Many additional proposals also being evaluated

Suggested criteria for C++0x

- General principles:
  - Minimize incompatibilities with C++98 and C99
  - Keep to the zero-overhead principle
  - Maintain or increase type safety
  - Minimize “implementation-defined” & “undefined”
- Core language goals:
  - Make C++ easier to teach and learn
  - Make rules more general and uniform
- Library goals:
  - Improve support for generic programming & other programming styles
  - Improve support for application areas

Kinds of issues being addressed

- Performance
- Selected specialized domain support
- Generalization, extension of current practice
- Component interoperability
- Coding convenience
- Improvements in type-safety, -correctness, and the type system itself

Features under consideration (partial list)

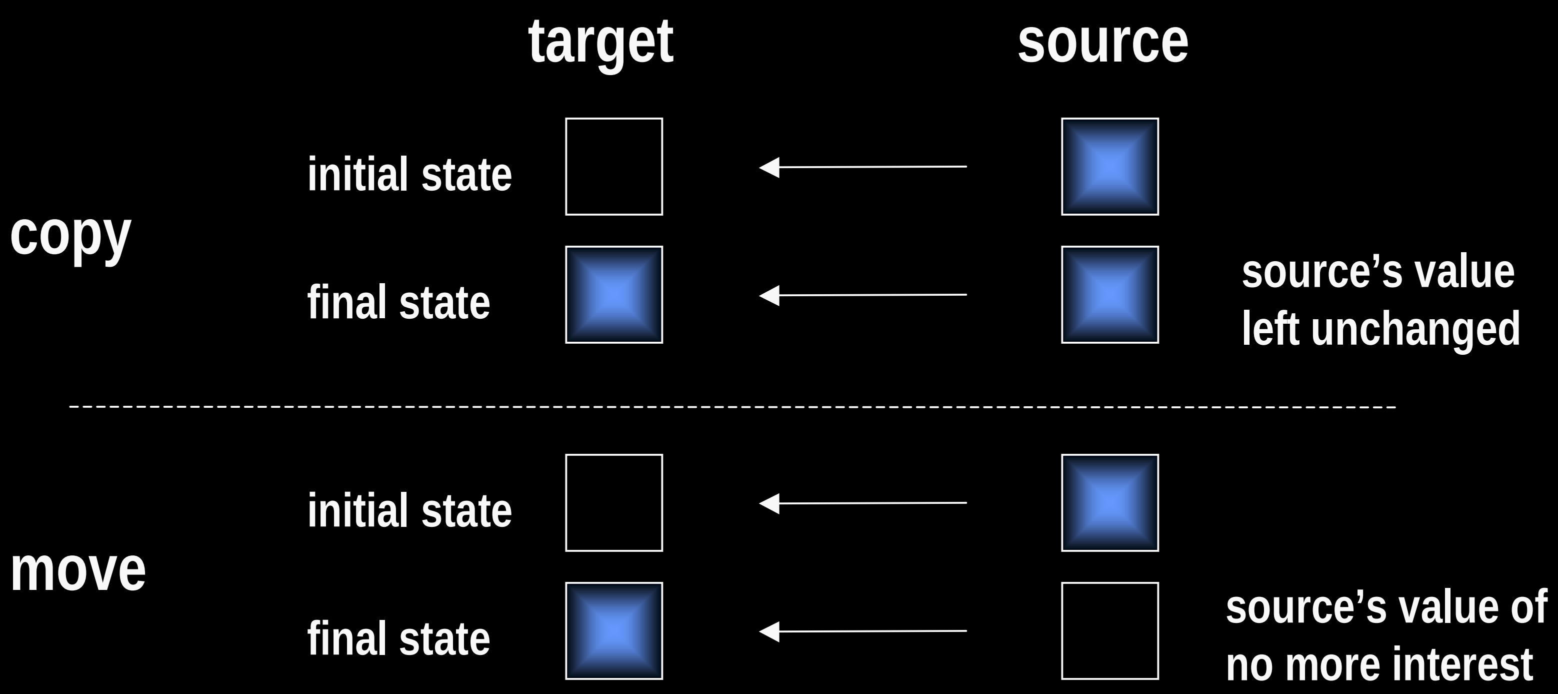
Core Language	Standard Library
Dynamic libraries	Random numbers
Move semantics	Mathematical special functions
Compile-time reflection	Shared-owner smart pointers
Concepts	Enhanced function binder
Static assertions	Unordered (hashed) containers
decltype and **auto**	Regular expressions
Forwarding constructors	Polymorphic fctn. obj. wrappers
Local classes as template parm's	Tuple types
User-defined literals	Type traits
Generalized initializer lists	Member pointer adaptors
Null pointer constant	Reference wrappers
Template aliases	Function result type traits

Issue: performance

- Representative proposal: *move semantics*
- Observation: copying an object can be expensive (*e.g.*, deep copies of linked storage structures)
- Basic idea: reduce cost, when possible, by *moving* instead of copying
- Typically possible when the source object:
  - Is disposable after the copy, or …
  - Is about to get a new value after the copy

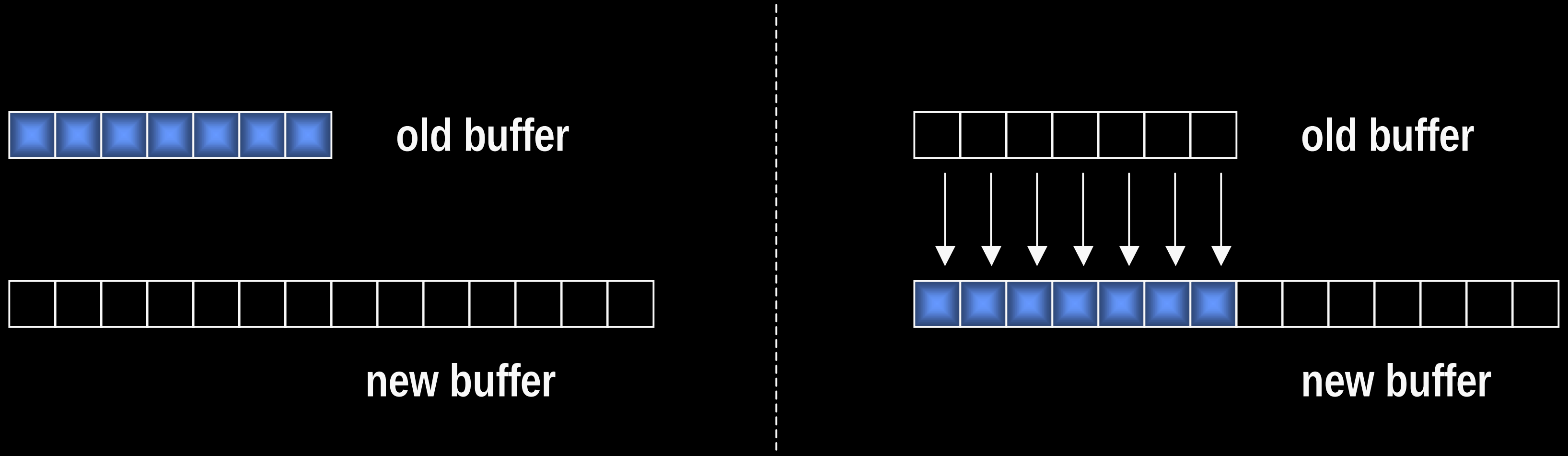
Move semantics (courtesy H. Hinnant)

- *Move* is the ability to cheaply transfer the value of an object from a source to a target, with no regard for the value of the source after the move:



Move-aware std::vector

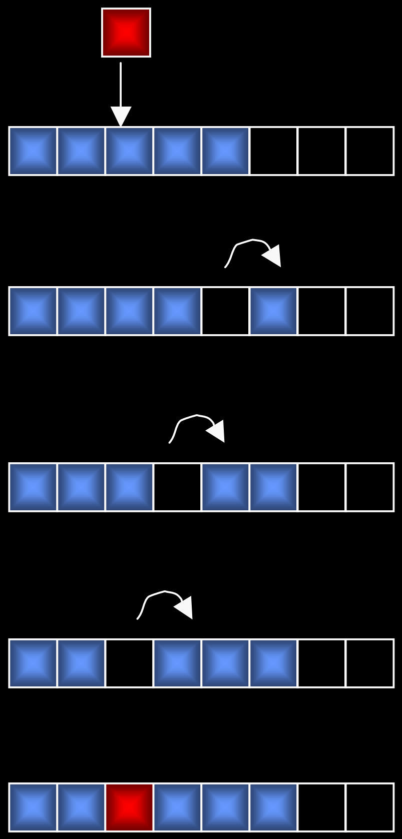
- **std::vector** can make good use of *move semantics* when creating a new internal buffer:



- Elements are *moved* (not copied) to the new buffer
- Since the entire old buffer is about to be destroyed, we don't care about its elements' post-move values

And further …

- **std::vector** can make good use of *move semantics* when inserting (or erasing) within a single buffer:



- Elements are *moved* (not copied) within the buffer to create a “hole” for the new element
- Since each “hole” soon receives a new value, we don’t care about its post-move value

Move semantics: timing examples

- **vector<string>::erase**

```
std::string  s( 20, ' ' );
std::vector<std::string>  v( 100, s );
clock_t  t  =  clock( );
v.erase( v.begin( ) );
t  =  clock( ) - t;
```

} ***Move semantics 14 times faster!***

- **vector<multiset<string> >::erase**

```
std::string  s( 20, ' ' );
std::multiset<std::string>  ms;
for  ( int i = 0;  i < 100;  ++I )
   ms.insert( s );
std::vector<std::multiset<std::string> >  v( 100, ms );
clock_t  t  =  clock( );
v.erase( v.begin( ) );
t  =  clock( ) - t;
```

} ***Move semantics 200 times faster!***

Issue: specialized domain support

- Representative proposals: *random numbers* and *mathematical special functions*
- Both of wide utility to scientific communities
- Current library support is minimal (**rand()** and trig functions), clearly inadequate for our applications
- Involves first significant enhancement to **<math.h>** in ~30 years

Features of *random numbers* proposal

- Design is based on a flexible and extensible framework:
  - It's easy to add user-defined distributions
  - Any such added distributions will work seamlessly with existing components
- Includes engines and distributions important to our community:
  - Engines' outputs are guaranteed to be portable and reproducible
  - Distributions' outputs are guaranteed to be reproducible

Summary of *random numbers* proposal

Engines	**Distributions**
Linear congruential	Uniform integer
Mersenne twister	Uniform floating-point
Subtract with carry	Binomial
Discard block	Exponential
Xor combine	Normal
	Gamma
	Poisson
	Geometric
	Bernoulli

Random numbers proposal status

- Fermilab hosted the proposal's author for a week in 2002 and provided design criteria and technical guidance
- Accepted for Library Technical Report
- Boost provides one near-implemention
- At least one vendor has implemented and is planning to ship
- We are proposing additional distributions for C++0x

Distributions approved and proposed

- "Uniform" family:
  - integer uniform, floating-point uniform
- "Bernoulli" family:
  - Bernoulli, binomial, geometric, negative binomial
- "Poisson" family:
  - Poisson, exponential, gamma, Weibull, extreme value
- "Normal" family:
  - Normal, lognormal, χ^2, Breit-Wigner, Fisher's *F*, Student's *t*
- "Sampling" family:
  - Histogram, cumulative distribution function
- "Addams" family:
  - (just kidding; sorry)

Summary of *special functions* proposal

- Bessel/Neumann (6)
- Legendre (2)
- Spherical harmonics
- Hermite
- Laguerre (2)
- Hypergeometric (2)
- Elliptic integrals (6)
- Beta
- Exponential integral
- Riemann zeta
- Error (2)
- Gamma

Why standardize *special functions?*

- Quality and reliability:
  - Professional attention to important details often overlooked by typical application programmers:
    - Lack of generality when a specific problem is at hand
    - Insufficient attention to details: corner cases, errors, …
- Portability and re-use:
  - Focus on problems rather than on issues related to infrastructure or platform dependency
- Significance:
  - Greatly enhance and promote usage among computing communities in the scientific, engineering, and mathematical disciplines

Special functions proposal status

- Initial reaction: reluctance by vendors, largely due to amount of work and perceived lack of general user interest
- Accepted for Library Technical Report as result of (ahem) our lobbying efforts
- Implementation by at least one vendor is well under way
- A bonus: also under active consideration for the C programming language

Issue: component interoperability

- Representative proposal: *shared-ownership smart (resource-managing) pointers*
- No pointer type having shared-ownership semantics is uniformly available today:
  - So we all reinvent and produce unique versions, a situation much like the days before **std::string**
  - Treated in depth by numerous textbooks, yet …
  - … correct smart-pointer implementation (even by experts) is known to be “exceedingly difficult” …
  - … and especially so when exceptions are taken into account

Example of a subtlety

```
class  C;

typedef   C *   C_ptr;

void  f( C_ptr, int );

int  g( );

void  oops( )   {
    C_ptr  p ( new C );
    f( p,  g( )  );          //  leaks memory if g( ) throws …
    delete  p;               //  … since we'll never get here
}
```

How shared_ptr<> helps

```
class  C;

typedef   shared_ptr<C>   C_ptr;

void  f( C_ptr, int );

int  g( );

void  okay( )   {
    C_ptr  p ( new C );
    f( p,  g( )  );          //  no leak, even if g( ) throws
    //  bonus: no client code need for explicit deletion
}
```

In brief

- Pointers naturally appear in function and library interfaces
- The only managing pointer in C++ today is **std::auto_ptr<>** but it has no shared ownership semantics
- Key insight: All information needed for proper managed object destruction is captured when a smart pointer is initialized

Features/benefits of shared_ptr<>

- Allows programmers to avoid pitfalls of:
  - Manual memory resource management
  - Memory access via dangling (invalid) pointer
- Provides:
  - Far clearer expression of programmer intent
  - Safer pointer parameter passage
- Has other uses and features:
  - Standard container contents (unlike **auto_ptr<>**)
  - Companion non-sharing observer **weak_ptr<>**
  - *Handle-body* and other pointer-based patterns and idioms

Issues: convenience, generalization

- Representative proposal: *enhanced function binder*
- Generalizes, extends current standard library adapters: **bind1st( )**, **bind2nd( )**, **ptr_fun( )**, **mem_fun( )**, **mem_fun_ref( )**
  - Applicable to functions, member functions, and function objects alike
  - Independent of arity
  - Well-suited for in-place use in conjunction with standard algorithms; often avoid need to code numerous out-of-line custom functions

Basics of bind

```
int  g( int a, int b )             {  return  a + b;  }

bind( g, 11, 12 )                  //  a niladic function object
bind( g, 11, 12 ) ( )              //  same as  g( 11, 12 )

bind( g, _1, 16 ) ( x )            //   equivalent to  g( x, 16 )
bind2nd(ptr_fun(g), 16)(x)              // g( x, 16 )


int  h( int a, int b, int c )      {  return  a + b + c;  }

bind( h, _3, _2, _1 ) ( x, y, z )       // h( z, y, x )
bind( h, _3, _3, _3 ) ( x, y, z )       // h( z, z, z )
```

Composition via bind

```
class  Track  {
    ...
    double  pT( )  const;
    double  dca( )  const;
};

std::vector< Track >  v( ... );

std:sort( v.begin( ), v.end( )
             , bind( less<double> ( )
                      , bind( & Track::pT,  _1 )
                      , bind( & Track::pT,  _2 )
             )        ) ;
```

“And now for something … different”

- Previous discussion focus:
  - Concrete proposals already accepted
  - Now being tweaked for final wording, *etc.*
- But there are many other ideas in various stages of discussion, development, drafting
- Of particular interest to our community:
  - Dynamic libraries (.so , .dll)
  - Reflection

Dynamic libraries

- "Components gathered together by the operating system when the application runs"
- Today "an application that uses dynamic libraries cannot be written entirely in standard C++"
- "The terminology, the compiler and linker mechanisms, and the semantic rules for dynamic libraries vary widely from system to system"

Important scenarios for dynamic libraries

- Library code that is provided via one or more dynamic libraries:
  - The C++ standard library
  - A third-party library
- Application code that uses one or more dynamic libraries:
  - All known at (static) link time
  - Explicitly loaded/unloaded at run time
  - Mixture of both?

Runtime linkage support issues

- Concepts and nomenclature not in the current Standard:
  - *“Linkage unit,” “linkage unit identifier,” “shared linkage,” “tentative resolution,”* …
- Runtime linkage impact on:
  - Program model & phases of translation
  - ODR (One-Definition Rule)
  - Type identification and other meta-data
  - Construction/destruction of **static** objects
- Declaration syntax describing runtime linkage
- Syntax/semantics of loadable libraries

Reflection

- Entities *reflect* when they examine themselves:
  - Can happen at compile time or at run time
  - Often expressed via a "meta-object protocol"
- Classical application is serialization for persistence:
  - Describing the object in some agnostic format
  - Many difficult issues: pointers, portability, …
  - Lots of library-based attempts, but limited success
  - Complete solution needs language support

Limited standardization activity to date

- Why?
  - Too many items competing for attention and resources
  - No agreed-upon “prior art” on which to standardize
- Research efforts under way:
  - EDG-based “Metacode” project (D. Vandevoorde)
  - gcc-based “Compile Time Reflection for C++” (G. Dos Reis, J. Maddock, *et al.*)
- We are writing a paper to try to spur Committee interest/activity

Sample of what else is on the horizon

- Computer arithmetic has historically been largely based on binary representation
- A recently-promulgated ISO standard promotes the cause of decimal arithmetic:
  - Primarily motivated by commercial interests, but also of interest to the scientific world
  - Vendor commitment to new hardware in support of decimal arithmetic
- Long-term view suggests:
  - Binary arithmetic will stagnate/fossilize, and
  - Decimal arithmetic will dominate numeric types

Moving forward on decimal arithmetic

- C++ is exploring language and library support for decimal arithmetic:
  - Historically unprecedented cooperation with ANSI and ISO Standards Committees for Programming Language C
  - Many thorny problems need to be addressed
- Sample of agenda:
  - New native decimal types
  - Supporting functionality (*e.g.*, operators, library functions, I/O, …)
  - Interoperability with binary data

Summary

- C++ continues to be of interest to Fermilab:
  - Expressiveness
  - Performance
  - Significant community experience
- C++ is being enhanced, along many axes, in directions of substantive interest to us:
  - We've been actively nudging it in these directions
- Standard components benefit us all:
  - Require less in-house development/maintenance
  - Enhance efforts to share code
  - Allow us to focus on physics, not infrastructure

References

- N1451: “A Case for Template Aliasing”
- N1452: “A Proposal to Add an Extensible Random Number Facility … (Revision 2)”
- N1542: “A Proposal to Add Mathematical *Special Functions* … (Version 3)”
- N1547: “Comments on the Initialization of Random Engines”
- N1588: “On Random-Number Distributions …”
- N1611: “*Implicitly-Callable Functions* …”
- Additional information

The C++ Standards Committee: Progress & Plans

February 17, 2004

Walter E. Brown Marc F. Paterno

Computing Division

Fermi National Accelerator Laboratory